



VLSI Memo No. 89-547  
July 1989

DTIC  
SELECTED  
SEP 05 1989  
S D  
Dcs

## Adaptive Backoff Synchronization Techniques

Anant Agarwal and Mathews Cherian

### Abstract

Shared-memory multiprocessors commonly use shared variables for synchronization. Our simulations of real parallel applications show that large-scale cache-coherent multiprocessors suffer significant amounts of invalidation traffic due to synchronization. Large multiprocessors that do not cache synchronization variables are often more severely impacted. If this synchronization traffic is not reduced or managed adequately, synchronization references can cause severe congestion in the network. We propose a class of adaptive backoff methods that do not use any extra hardware and can significantly reduce the memory traffic to synchronization variables. These methods use synchronization state to reduce polling of synchronization variables. Our simulations show that when the number of processors participating in a barrier synchronization is small compared to the time of arrival of the processors, reductions of 20 percent to over 95 percent in synchronization traffic can be achieved at no extra cost. In other situations adaptive backoff techniques result in a tradeoff between reduced network accesses and increased processor idle time.

DISTRIBUTION STATEMENT A  
Approved for public release  
Distribution Unlimited

89 9 01 026

### Acknowledgements

To appear in the *16th Annual Symposium on Computer Architecture*, Jerusalem, Israel, June 1989. This research was supported in part by the Defense Advanced Research Projects Agency under contract N00014-87-K-0825 and by IBM under a joint research program.

### Author Information

Agarwal: Laboratory for Computer Science, Room NE43-418, MIT, Cambridge, MA 02139. (617) 253-1448.

Cherian, *current address*: Intel Corporation, Santa Clara, CA 95051.

Copyright© 1989 MIT. Memos in this series are for use inside MIT and are not considered to be published merely by virtue of appearing in this series. This copy is for private circulation only and may not be further copied or distributed, except for government purposes, if the paper acknowledges U. S. Government sponsorship. References to this work should be either to the published version, if any, or in the form "private communication." For information about the ideas expressed herein, contact the author directly. For information about this series, contact Microsystems Research Center, Room 39-321, MIT, Cambridge, MA 02139; (617) 253-8138.

# Adaptive Backoff Synchronization Techniques<sup>1</sup>

Anant Agarwal and Mathews Cherian  
Laboratory for Computer Science  
Massachusetts Institute of Technology  
Cambridge, MA 02139

## Abstract

Shared-memory multiprocessors commonly use shared variables for synchronization. Our simulations of real parallel applications show that large-scale cache-coherent multiprocessors suffer significant amounts of invalidation traffic due to synchronization. Large multiprocessors that do not cache synchronization variables are often more severely impacted. If this synchronization traffic is not reduced or managed adequately, synchronization references can cause severe congestion in the network. We propose a class of adaptive backoff methods that do not use any extra hardware and can significantly reduce the memory traffic to synchronization variables. These methods use synchronization state to reduce polling of synchronization variables. Our simulations show that when the number of processors participating in a barrier synchronization is small compared to the time of arrival of the processors, reductions of 20 percent to over 95 percent in synchronization traffic can be achieved at no extra cost. In other situations adaptive backoff techniques result in a tradeoff between reduced network accesses and increased processor idle time.

Accession For	
NTIS	CRA&I <input checked="" type="checkbox"/>
DTIC	TAB <input type="checkbox"/>
Unannounced <input type="checkbox"/>	
Justification	
By <u>per Ltr</u>	
Distribution /	
Availability Codes	
Dist	Avail and/or Special
A-1	

<sup>1</sup>Appears in 18th Annual Symposium on Computer Architecture, Jerusalem, Israel, June 1989.



# 1 Introduction

Processor self-scheduling schemes in shared-memory multiprocessors commonly use shared variables to synchronize activities among processors [6, 22, 15]. This use of synchronization variables often leads to widespread sharing among processors. Our trace-driven simulations of parallel applications show that these widely shared synchronization variables adversely impact the performance of large-scale multiprocessors, cache-coherent or otherwise.

In systems without hardware support for cache coherence, such as the IBM RP3 [18], Ultracomputer [9], Cedar [7], these references to shared variables must traverse the interconnection network. Not only do synchronization references consume a significant fraction of the network bandwidth, but more important, a widely-shared synchronization variable (such as in a barrier synchronization) will result in heavy traffic to the same location in memory and cause hot-spot contention problems [19].

On the other hand, in systems that use directory schemes to maintain cache coherence, we show that synchronization variables result in excessive invalidation traffic when the number of pointers in the cache directory is limited. A potential solution for the cache directories would be to implement software combining trees [25] for synchronization variables. As long as the degree of the nodes in the combining tree is less than the number of pointers in the cache-directory, then synchronization variables will not result in extra invalidation traffic. We are currently investigating this approach and will not address it here. An alternate method is to disallow caching of synchronization variables.

In this paper we consider software schemes to reduce the number of synchronization spins in multiprocessors that do not cache their synchronization variables. We propose a set of adaptive backoff techniques which make use of available synchronization state information in order to "back off" and postpone polling a synchronization variable.

The general idea of backoff has been used in one form or another in a number of applications. The approach was first used in Aloha [1], a radio-based, packet-switching network. If a collision occurred in the network, each source would backoff for a random interval before attempting to retransmit. The Ethernet [16] went one step further and used a random retransmission interval in which collision history influenced the choice of the mean of the random intervals. Adaptive control schemes for multiple access communications networks have been analyzed in [13, 12, 14]. In addition to backoff history, we use information such as the expected time that the resource becomes available, or the network load, and adapt to the current circumstances.

We evaluate the performance of adaptive backoff synchronization techniques by applying them to the barrier synchronization. Barrier synchronizations are commonly used in applications to guarantee that all processors have reached a point in a program before proceeding.

This paper focuses on barriers implemented using two shared variables with busy waiting (or spinning) on synchronization variables [22] (described in detail in a later section). While this form of implementation is quite common, especially when exploiting fine-grain parallelism, alternate barrier implementations might use a scheme where all but the last processor to arrive at the barrier are put to sleep (or blocked). Reactivation of the processors is contingent on a condition variable signalled when the last process arrives at the barrier. This method avoids the extra network traffic of polling a barrier flag, but incurs the potentially high overhead of enqueueing a process on a condition variable. Often, the choice of busy waiting or blocking cannot be made at compile time due to uncertainty in execution times of processes. In such cases, our adaptive methods can be used to decide when it might be best to take a busy-waiting process out of circulation and queue it on a condition variable as explained in a later section.

Hardware support for barriers has also been proposed in several forms. The RP3 [18] proposed using a combining network in which the switches contain special hardware to combine simultaneous data accesses destined to the same location in memory and forward one request. This would eliminate contention in the network and at the memory modules, but RP3 cost estimates for this approach predict that switch size and/or cost for a 2 X 2 switch could increase by a factor between 6 and 32. Several cache-coherent multiprocessors allow simultaneous invalidates of all cached copies of a block. In such systems all repeat accesses of a synchronization variable can be satisfied by the cache. However, the need to rely on resources that can support broadcast invalidates, such as a shared bus, limits the scalability of such systems. The PAX computer [10] uses special global-synchronization logic implemented in hardware to allow low-latency, low-cost barrier synchronization.

Issues which arise with this approach concern flexibility in allowing multiple numbers of barriers to execute simultaneously with varying numbers of processors.

Our results show that backoff techniques applied to barriers yield reductions in synchronization traffic by 20 percent to over 95 percent in cases where the number of processors involved in the barrier is small compared to the time of arrival between processors. In other situations, these schemes provide a tradeoff between cost (in terms of processor idle time) and performance. The user can determine this tradeoff depending on particular needs or the application being run. We also discuss other applications of adaptive backoff schemes in Section 8.

The rest of this paper is organized as follows. We first present results from our trace-driven simulations describing how synchronization impacts large-scale multiprocessors. We then describe the network model that we assume for this study. Section 4 presents the adaptive backoff synchronization techniques as they apply to barriers. We then discuss the barrier evaluation model and our simulation methodology. We evaluate these ideas and discuss the tradeoffs involved in their implementation using a simple analytical model and through simulations in Sections 6 and 7. Sections 8 and 9 suggest extensions to our work and summarize our findings.

## 2 The Synchronization Problem

In this section we present data from trace-driven simulations of the FFT [4], SIMPLE [5], and WEATHER [11] applications and explain why synchronization is a problem in large-scale systems. We will illustrate the problem through the barrier synchronization example.

A typical implementation of a barrier might use a shared variable whose initial value is zero. Each processor arriving at the barrier increments the shared variable. If the variable attains the value  $N$ , implying that all  $N$  processors have reached the barrier, the processor can proceed. Otherwise, it repeatedly tests the barrier until the above condition is true. The increment operation on the barrier variable must be atomic. This implementation has the drawback that each processor attempting to increment the barrier variable must contend with all the others simply polling it to test for the proceed condition.

A better implementation, e.g., Tang and Yew's [22], splits the barrier into two shared variables: an incrementing variable (henceforth called the barrier variable) initially set to zero, and a barrier flag variable also initially reset. An arriving processor increments the barrier variable. If the variable's value is less than  $N$ , the processor polls the barrier flag which is set by the last processor to reach the barrier. Even this scheme requires that the last processor to reach the barrier compete with the  $N-1$  processors testing the barrier flag when it tries to set the flag.

The important point to note, however, is that in both implementations, the shared variables involved are necessarily shared among all processors in the system. It is precisely this widespread sharing which impacts performance when scaling to large systems.

### 2.1 Synchronization References and Scalability

The widespread sharing that occurs with synchronization variables is not a problem when used in bus-based snoopy-cache multiprocessors [8, 23]. Because snoopy-cache-based protocols perform broadcast invalidates or updates, a variable shared among all processors generates no more traffic on the shared bus than a variable shared among only two processors. The limitation of snoopy-based schemes, however, is that they do not scale to large multiprocessor systems. Since these schemes require low latency broadcasts for cache coherence, as well as the ability to "watch" all bus transactions, they must use a shared bus for communication. A single bus cannot offer the bandwidth demanded by large-scale shared-memory multiprocessors.

Unfortunately widespread sharing of synchronization variables can drastically impair performance in large-scale multiprocessors, cache-coherent or otherwise. First, let us consider multiprocessors with coherent caches, where a directory is used to keep track of cached copies of shared blocks. In general, for every memory block, a directory must store as many pointers as the number of processors (say  $N$ ) in the system [3]. Such a scheme is termed  $Dir_NNB$ , for  $N$ -pointers-No-Broadcast in [2]. In practice, it is possible to maintain just  $i$  pointers ( $i < N$ ) to yield the  $Dir_iNB$  scheme [2]. Invalidations are forced to limit the cached copies of a block to  $i$ , or to gain exclusive ownership on a write. Results in [2] showed that during an invalidation situation, few invalidations

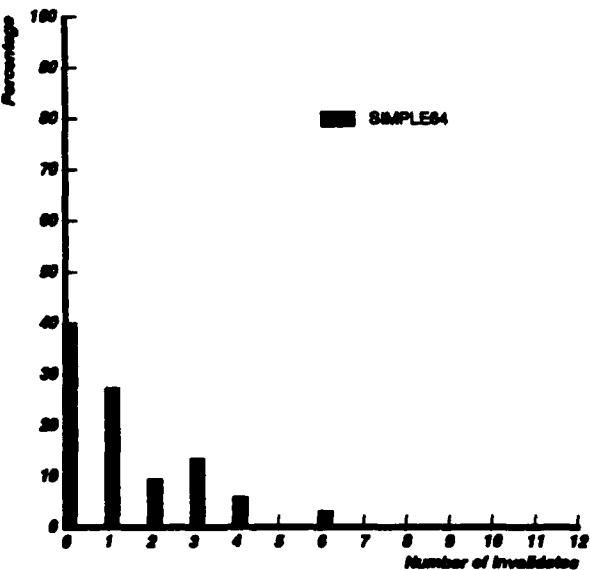


Figure 1: Cache invalidation statistics for SIMPLE with 64 processors. The height of a bar at  $x$  reflects the fraction of write hits to previously clean blocks that resulted in  $x$  invalidation messages.

were actually necessary. Results from our trace-driven simulations of 64-processor systems discussed below as well as the results in [24] corroborate the findings in [2].

Figure 1 shows an invalidation histogram for a 64-processor simulation of  $Dir_NNB$  driven by a trace from the SIMPLE application. We also ran simulations on FFT and WEATHER application traces with 64 processors.<sup>2</sup> The simulations used direct-mapped caches of size 256KBytes and block size 16 bytes. The graph shows the histogram of the number of invalidations required during a write to a previously clean block. We see that in over 95 percent of the times that an invalidation occurred (in both 16 and 64 processor simulations), a block had to be invalidated from no more than three caches. Invalidation histograms for FFT and WEATHER had a corresponding figure of over 99 percent. The graphs shows the percentage of writes which resulted in invalidations to up to 12 caches. Writes resulting in invalidations of greater numbers of caches were proportionately insignificant.

Why do synchronization references hurt performance? Our simulations revealed that synchronization variables were largely responsible for the cases in which more than three caches were invalidated. Synchronization references are even more damaging when the effect of simultaneous read sharing is considered. Recall that using  $i$  pointers limits simultaneous read sharing of a block to only  $i$  copies, and invalidations must occur to enforce this rule. For synchronizations like barriers, active sharing might occur among all processors involved, resulting in a high invalidation rate in directory-based schemes.

Table 1 shows the fraction of synchronization references out of the total number of synchronization references which resulted in an invalidation. The percentage is far higher than the corresponding fraction for non-synchronization data references. The values in the table are slightly pessimistic, because the processors were simulated to make memory requests in round robin fashion (see Section A in the Appendix for more details). In all cases the percentages of references resulting in invalidations for both non-synchronization and synchronization references improves as the number of pointers in the scheme increases from two to three.

It is clear that invalidation traffic due to synchronizations can be deleterious to the performance of cache-coherent multiprocessors. One solution is to use software combining trees. Alternatively, one can disallow

<sup>2</sup>See Section A in the Appendix for a description of the applications, the tracing technique, and multiprocessor simulation methodology.

Application	Pointers	Non-Synch.	Synch.
SIMPLE	2	8.5	93.5
	3	7.1	81.3
	4	6.0	81.1
	5	5.2	99.9
	64	.53	1.2
WEATHER	2	1.9	99.9
	3	1.7	99.9
	4	1.5	99.9
	5	1.5	99.9
	64	1.2	.03
FFT	2	6.7	99.0
	3	5.0	99.0
	4	3.5	98.9
	5	3.5	98.8
	64	3.5	3.5

Table 1: Percentage of synchronization and non-synchronization references that cause invalidations in directory schemes with 2, 3, 4, 5, and 64 pointers. Synchronization references comprised 0.2%, 7.9%, and 5.3% of the data references in FFT, WEATHER, and SIMPLE respectively.

caching synchronization variables.

## 2.2 Disallowing Caching of Synchronization Variables

If most synchronization accesses cause invalidations that involve multiword transfers, then why cache synchronization variables? The problems with this approach are similar to those in multiprocessors that make all shared locations uncacheable: increased network traffic and potential hot-spot contention. Synchronization references, such as those due to a barrier, are often to the same location in memory and only a small percentage of all data accesses to the same "hot" module can cause tree saturation [19] in the interconnection network and a corresponding severe drop in the effective memory bandwidth.

Table 2 shows that the percentage of uncached synchronization traffic to memory out of the total data traffic can be large. We compute traffic to memory by summing the total number of network transactions generated by references. For example, in the case of a cache miss, two network transactions are generated: one to send the requested address to memory and one to send the requested data from memory to the processor.

The reason SIMPLE and WEATHER generate far more synchronization traffic than FFT is that their load balancing is not as good as in FFT (see Section A for details), resulting in more synchronization accesses at loop barriers as processors wait for all processors to arrive. The slight relative increase of synchronization overhead in all cases when going from two to five pointers is because synchronization traffic remained constant while invalidation traffic (part of total memory traffic) decreased as more pointers were available for sharing of blocks.

Therefore, if we are to scale multiprocessors, network traffic due to synchronization must be rigorously minimized.

In large-scale shared-memory multiprocessors, such as the RP3, Ultracomputer, Cedar, all traffic to shared variables must go over the network<sup>3</sup>, and the relative fraction of network accesses attributable to synchronization is slightly smaller. We measured memory traffic when shared variables were not cached and found that synchronization traffic accounted for 25.5%, 49.2%, and 1.47% of the total traffic in SIMPLE, WEATHER, and FFT, respectively. Our motivation for reducing the network traffic, especially traffic that is partial to a specific memory location, still remains.

The adaptive backoff techniques we are proposing are software solutions to help alleviate the hot-spot contention problem by reducing the number of idle synchronization spins. These techniques could even be used

<sup>3</sup>Although temporary caching of shared locations with compiler inserted cache flush directives can help relieve network load.

Application	Pointers	Traffic (%)
SIMPLE	2	22.0
	3	23.5
	4	24.6
	5	25.8
	64	35.3
WEATHER	2	55.4
	3	56.3
	4	57.4
	5	57.6
	64	59.9
FFT	2	1.3
	3	1.4
	4	1.5
	5	1.5
	64	1.5

Table 2: Synchronisation traffic to main memory as a percentage of the total traffic when the synchronisation variables are not cached. Block size is assumed to be 16 bytes and cache size is 256KBytes. The non-synchronization blocks are cached and coherence is maintained using directory schemes with 2, 3, 4, 5, and 64 pointers.

in conjunction with hardware solutions such as a combining network. The combining network is much slower than a conventional network, so we still would like to reduce the amount of synchronization traffic traversing the network.

### 3 The Network Model

The network model that we assume is the following: processors can access any memory over the network in one network cycle. We do not model network contention, but do model contention for the barrier variable and flag. We also assume that the barrier variable and flag are in different memory modules, so simultaneous requests to the two by different processors can be satisfied. We assume that in a network cycle only one processor can access the barrier variable or the barrier flag. If a processor is denied access to the variable in a network cycle it repeats the access to the variable in the next network cycle. This model might correspond to a crossbar switch where the only contention is for the end memory modules that have the barrier variable and flag; contention due to other non-synchronization references is not included. It also roughly approximates the performance of a circuit-switched multistage interconnection network, where the network cycle time can be the round-trip time over the network. In the latter case the contention at intermediate network nodes is not included.

The network traffic rates computed using our barrier scheme might also be input into a more complex model of a multistage interconnection network such as that proposed by Patel [17] if network contention results are desired. Unfortunately Patel's model does not account for hot-spot contention. We are also using large parallel traces of real applications derived using various synchronization schemes to drive network models to obtain performance estimates in the presence of hot-spots caused by barrier traffic and when the barrier traffic is reduced using our techniques.

### 4 Adaptive Backoff Barrier Synchronization

The basic idea behind adaptive backoff methods is simple. An adaptive backoff barrier technique makes use of available information in deciding how long to wait before trying to read a barrier flag rather than continuously polling the flag. If necessary, the adaptive method can also provide a hint to the processor to queue itself on the barrier flag.

We will assume barriers implemented using a separate barrier variable and a barrier flag as described earlier.

If the barrier variable and flag are one and the same object, the relative advantage of using adaptive backoff techniques will be even greater.

#### 4.1 Backoff on the barrier variable

The first method, called backoff on the barrier variable, is the simplest and tries to reduce unnecessary network accesses on the barrier flag. In this method, the barrier implementation is optimized by making use of the state of the barrier variable. The barrier variable value reveals the number of processors waiting at the barrier. Let there be  $N$  processors that must arrive at the barrier, and let the average memory access time over the network be 1 cycle as mentioned earlier. If  $i$  processors have reached the barrier, then an arriving processor can start polling the barrier flag at least  $(N-i)$  cycles after reaching the barrier variable  $A$ . Waiting to re-poll the barrier variable can be implemented as a processor loop that does not access memory, with the loop count set as function of the waiting time.

#### 4.2 Backoff on the barrier flag

We will also look at other methods that try to further reduce the number of spins on the barrier flag. Processors can keep track of the number of times they have polled the barrier flag and correspondingly backoff by a linear or exponential amount the longer they have waited. This code can be part of the barrier implementation in software and needs no hardware support. We call this group of techniques backoff on the barrier flag. In all our discussions of the performance of these latter methods, we assume that backoff on the barrier variable is also applied.

In backoff on the barrier variable, if the interarrival times of processors are very large, then a processor might wait its  $N-i$  cycles and start polling the barrier flag long before the last processor arrives at the barrier. In these situations, we might wait longer before polling the flag, say  $(N-i)+C$  or  $(N-i)*C$ , where  $C$  is some positive integer. While this might reduce the number of unnecessary network accesses, it might also cause the processor to remain idle and miss accessing the barrier at the earliest it becomes available. We suggest some methods of choosing appropriate backoff parameters in Section 8.

In backoff on the barrier flag, there exists a danger of backing off much more than necessary. Clearly there is a tradeoff between network access reduction and cpu idle time. If only a few processors are involved in a barrier synchronization, then to reduce the hot-spot contention problem, one might prefer to take the hit in cpu idle time for these contending processors so that the remaining processors in the system can perform unhindered. As mentioned before, even a small percentage of memory references to the same "hot" memory module can result in severe congestion of the interconnection network, thereby reducing all processors' utilization [19]. Of course, if all processors in a system are involved in a barrier synchronization, then the cpu idle time becomes an important consideration.

Note that the backoff algorithm we use is deterministic, unlike the adaptive control algorithms used in [13, 12, 14] where the probability of a retry is adaptively adjusted. We choose this route for the following reasons: 1) we want backoffs to be as efficient as possible. Our deterministic backoff can be computed in a few instructions as opposed to the hundreds of instructions which would be necessary to compute retry probabilities adaptively and determine whether or not to perform a retry every cycle; and 2) often when processors first contend for a synchronization variable such as a barrier flag, their execution becomes serialized. Once serialized, the processors experience no contention the next time they poll the barrier flag. Since all the processors backoff by equal amounts the serialization is preserved. However, if the processors retry probabilistically, the serialization is destroyed and could result in contention again.

Backoff decisions are made only when a process has just updated the barrier variable, and when the process has read the barrier flag and the flag is not set. So, once a processor initiates a barrier read request, the network controller for that processor attempts to read the barrier. If contention thwarts this attempt, the access is repeated until the flag is read. We do propose some other schemes where the network controller can back off if the congestion in the network is high.

For software-tree based implementations of barriers on non-cache-coherent multiprocessor as suggested by Yew, Tzeng, and Lawrie [25], our methods can still be used to reduce the spins on the intermediate nodes of



Figure 2: Intervals of execution and synchronization.

the tree.

We evaluate these ideas using a barrier model through analysis and simulations and discuss the tradeoffs between reduced synchronization accesses and wasted cpu cycles.

## 5 A Barrier Model

We will first describe the model that we use to evaluate barriers. We use two metrics: (1) the number of network accesses per process in accessing the barrier variable and barrier flag; and (2) the number of cycles that an average process spends from the time it arrives at the barrier to the time it is allowed to proceed from the barrier.

Overall performance is impacted by the total network traffic, which includes the regular non-barrier traffic and the barrier traffic. Because we currently do not model hot-spot traffic contention in the network, we preferred to present the numbers for the barrier traffic alone, as average numbers for overall traffic might be misleading in terms of the adverse effect of the barrier traffic focused on one memory module. We also provide measurements of the time between barrier accesses in parallel applications. If necessary our barrier traffic numbers can be amortized over this entire period to get the contribution of barrier traffic to overall average traffic.

Let us define  $A$  to be the time interval during which processes can arrive at the barrier.  $A$  is the time from the first processor's arrival at the barrier variable to the last processor's arrival at the barrier variable. The complementary interval between these two events we call  $E$ , i.e., the time between barriers in an application. If we were to follow an application's execution through time,  $E$  and  $A$  would appear as shown in Figure 2.

We measured  $A$  for our three applications. In Table 3,  $A$  is defined to be the number of cpu cycles from the time the first processor starts polling the barrier flag to the time the last processor sets the barrier flag. It is interesting to note that the average  $A$  for SIMPLE and WEATHER did not increase as greatly as for FFT when going from 16 to 64 processors. For highly uniform and load-balanced applications such as FFT the spread among arrivals is primarily due to the serialization which takes place at the loop index assignment. Thus, FFT was relatively more affected than the other applications when the number of processors increased.

The reason  $E$  and  $A$  for SIMPLE and WEATHER with 64 processors are similarly sized intervals is because the applications were not perfectly load-balanced. Not all the parallel loops contained a nice multiple of iterations which could be distributed evenly among all processors. The few processors who did not get work went straight to the barrier at the end of the loop.

The barrier model that we use for our analysis and simulations is actually slightly different and allows us to model a varying number of synchronizing processors for a given value of  $A$ . Our measurements of  $A$  from the applications were for a relatively large number of processors and this measurement yields an indication of the maximum time span between the first and last arrival at a synchronization point in that application. It is likely that a smaller number of processors can have an actual value of  $A$  much smaller than this maximum span. Therefore, we now define  $A$  to be the interval during which processors *may* arrive at the barrier, and  $N$  to be the number of synchronizing processors. We further assume that each processor has a uniform probability of appearing at any time instant during the interval  $A$ . From the uniform probability of arrival during the interval  $A$  we must compute the average time span between the first and last arrivals out of a total of  $N$  arrivals. This span must tend to  $A$  as  $N$  becomes large.

Application	Processors	A	E
SIMPLE	16	7021	42007
	64	7068	6195
WEATHER	16	82754	495298
	64	82787	82716
FFT	16	237	228073
	64	285	57997

Table 3: Average number of cycles, A, between first and last arrivals at waits and barriers. E is the average number of cycles between the last arrival at the previous barrier (or wait) and the first arrival at the next barrier (or wait), i.e. it is the average time between barriers or waits.

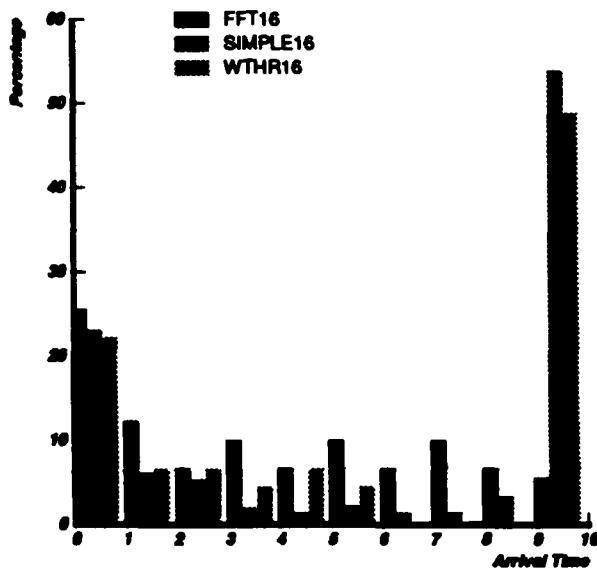


Figure 3: Arrival distribution of the processors involved in a synchronisation during the interval A.

To determine whether our assumption of uniform probability of arrival within A was reasonable we measured the arrival times in our applications and plot the times in a histogram in Figure 3. It is easy to see that the distribution is roughly uniform for FFT but is skewed towards the beginning and the end of the interval for SIMPLE. This skewing occurs because of uneven load- balancing. We observed, however, in the last peak that processor arrivals were still uniform over the last 200 references. There seems to be no real pattern and our assumption of a uniform distribution is not expected to significantly change our results for minor variations in the arrivals. We also present additional validation of this model by comparing the predictions obtained through simulations using the model and through measurements using the actual traces in Section 7.1.

### 5.1 Analytically Estimating Barrier Performance

We first present some simple calculations for extreme cases of A to determine the bounds on the possible savings and to provide insight into our simulations.

For the case  $A = 0$  (all processors arrive simultaneously) and no backoff, a processor will make on average  $N + N + N/2$  synchronization references. Each processor makes on average  $N/2$  references to get at the barrier variable, polls the barrier flag  $N/2$  references before the last processor gets through the barrier variable, continues polling the barrier flag  $N$  times until the last processor can set the flag, and finally leaves after  $N/2$

references, on average. We denote this model that assumes arrival at the same instant as Model 1.

If  $A \gg N$ , there is practically no contention to get the barrier variable. In this case we assume that processors appear at the barrier at a given time instant within the time interval  $A$  with uniform probability. Let us first compute the average time span  $\tau$  between the first and the last arrival within the interval  $A$  given  $N$  processors. The average time from the beginning of the interval to the first arrival can be shown to be  $A/(N+1)$ , and the average time from the last arrival to the end of the interval to be  $AN/(N+1)$ . The required time span  $\tau$  is the difference of the two, or

$$\tau = A \frac{N-1}{N+1} \quad (1)$$

Observe that  $\tau$  approaches  $A$  as  $N$  becomes large. Thus, each processor make on average  $\tau/2 + N + N/2$  network accesses during the synchronization phase. We call this Model 2.

Let us now consider backoff on the barrier variable. In this technique, we backoff an amount proportional to the value of the barrier variable. If  $i$  is the value of the barrier variable upon a processor's arrival, then the processor can wait  $N-i$  cycles before beginning to poll the barrier flag. When  $A = 0$ , the average number of synchronization accesses becomes  $N/2 + N + N/2$  cycles because the processor does not start polling the flag until the last processor gets through the barrier. A similar savings of  $N/2$  is made for  $A \gg N$ . With backoff only on the barrier variable, the potential savings get smaller as  $A$  gets larger because the savings is a constant  $N/2$  no matter what  $A$  is.

Of course, a modified scheme that backs off some constant factor times the value in the barrier to account for the non-unit time cost of accessing the barrier value, will provide a higher savings in network traffic, but it also adds the potential of increasing cpu idle time. We still have more state information we can use in the barrier: the number of times the barrier flag has been polled.

Rather than continuously polling the barrier flag until it is set, we backoff by some function of the number of times we have already read the shared variable. Backoff on the barrier flag is especially useful when  $A > N$ . In addition it can also help prevent interference with the final processor write request that will release the processes waiting on the flag. From Model 2 for  $A \gg N$  presented earlier, the potential savings in network accesses can be as large as  $\log_b(\tau/2)$  for exponential backoff, where  $b$  is the basis of the exponential backoff algorithm used. The backoff on the barrier flag can incur a high penalty – we might backoff too far, and waste cpu cycles. This idea is tested out in simulations which are discussed in the next section.

Finally, we present some network access rates for barriers on multiprocessors with hardware support for barrier synchronization to provide a basis for comparison with the backoff schemes. Examples of such hardware support are a bus to allow global invalidations (or global update) of cache entries, a directory with a full pointer map, and special logic to implement a global synchronization gate [10]. If there are  $n$  processors the invalidating bus incurs  $3n+1$  accesses for a barrier,  $n$  fetches of the barrier variable,  $n$  invalidations for  $n$  writes of the barrier variable,  $n$  fetches of the flag, and the final global invalidation caused by the write into the barrier flag, yielding roughly 3 accesses per processor per barrier operation. The updating bus (or an invalidating scheme that can detect a fetch with intent to write) would use  $n$  less than the previous scheme for roughly 2 bus accesses per processor. Like the bus, the directory scheme must incur  $3n$  on barrier variable accesses and invalidations, and flag accesses, but lacking a global broadcast must incur an additional  $n$  for the individual invalidates on the final write to the barrier flag, yielding 4 on average per processor per barrier operation. The Hoshino scheme uses  $n$  accesses to the global synchronization gate and the final single broadcast message to the participants to inform them to proceed, for a per-processor average of 1.

## 5.2 Simulation Methodology

We also use simulations to predict barrier performance with and without backoff. The barrier and network models are the same as described previously. Our simulation methodology is described here.

In our simulations we set a value for  $A$  and simulated processors arriving with uniform probability during this interval. Each processor first increments the barrier variable and then spins on the barrier flag until it is set by the last arriving processor. Our previous data in Table 3 showed that for three applications the value of  $E$  was between 6195 and 495298 cycles on average and the value of  $A$  was between 237 and 82787 cycles.

Clearly a wide range is possible and so we simulated  $A$  with a wide range and we will show the results for  $A = 0, 100, 1000$  for brevity. The important factor here is the relative size of the interval to the number of processors involved in the barrier - as our results will show. We chose  $A$ 's which span the entire spectrum.

Each simulation run measured the average number of network accesses made by a process from the time it arrived at the barrier variable to the time it proceeded from the barrier flag after having successfully tested the flag and observing a true value. As mentioned before, the number of network accesses includes contention for the barrier. We also measured the average time each process spent from the time it arrived at the barrier to the time it left.

The simulation for each set of parameters is repeated 100 times and the numbers are averaged over all the runs to compensate for the random variations due to the assumption of a uniform probability of arrival. We verified that for each of the numbers we present the standard deviation was less than about 7% over the hundred runs.

## 6 Evaluation

We evaluate the backoff methods using the models just described. This section first compares the predictions of the model with simulations. We then estimate the potential savings in network traffic using backoff techniques and discuss the tradeoffs involved in choosing the right parameters for the backoff algorithm.

### 6.1 Estimating the Potential Reduction in Traffic Using Analysis

We will first analyze the accuracy of our simple model in predicting the behavior of the barrier synchronization under various load conditions. The model will indicate the range of performance gains that we might expect using the backoff techniques and give insight into our simulation numbers.

In Figure 4 we compare the curves predicted by our model with simulation results and display the predicted network accesses for three cases:  $A = 0, A = 100, A = 1000$ . We will only compare the non-backoff performance for validation. The model can be modified to predict the performance of the backoff schemes, but for certain cases it can get quite complicated. We will, however, mention what terms in the model equations get impacted by the various schemes.

The network accesses for  $A = 0, A = 100$  do not differ much overall, but the way in which they differ is significant. For  $N < 32$ ,  $A = 0$  results in fewer accesses than  $A = 100$  because when  $A = 0$  processes do not have to wait for the last processor to arrive at the barrier. For larger  $N$ , however,  $A = 100$  starts performing better because when the arrivals are spread out slightly, there is less contention in accessing the barrier. We observe a similar behavior for  $A = 1000$  as  $N$  approaches  $A$ . As expected when  $N$  is small,  $A = 1000$  makes far more accesses than  $A = 0$  or  $A = 100$ .

The model is accurate as the figure shows. Model 1, as expected matches the curves for the  $A \ll N$  cases. In particular, Model 1 closely approximates the  $A = 0$  case, and yields a good match with the  $A = 100$  curve for  $N > 16$ .

Model 2 matches all the cases where  $A \gg N$ . Specifically, the Model 2 curve for  $A = 1000$  provides a near perfect match with the corresponding simulation curve for all the values of  $N$  shown. The Model 2 curve for  $A = 100$  matches the simulation  $A = 100$  curve for  $N < 128$ . When  $N$  is greater than 128, the model begins to underestimate the contention in accessing the barrier variable. In general, the maximum of the predictions of the two models yield a good fit with simulation in all ranges.

The model implies that for the case where  $N > A$ , the potential reduction in network traffic is 20%. When  $A > N$ , the potential gains are much more significant. If an exponential backoff method is used with constant  $e$ , then if the network accesses of the flag were  $M$ , with backoff these accesses can be reduced to the order of  $\log_e(M)$ . Because the waiting processes are not busily accessing the flag, the final process that must set the flag can usually proceed to update the flag without contending with the other processes.

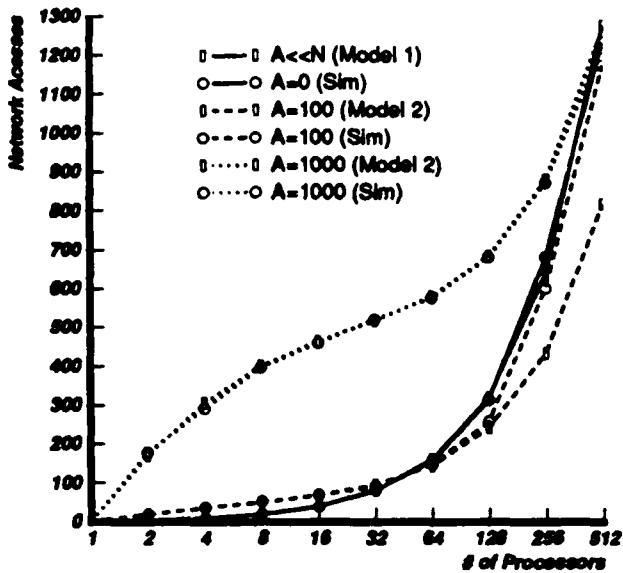


Figure 4: Comparing the predictions of the analytical model and predictions of barrier performance.

## 6.2 Simulation results

We now present simulation results for barrier synchronization performance. Figure 5 shows the net accesses for  $N$  ranging from 2 through 512 when  $A = 0$ , i.e., when all processes arrive at the barrier at the same time. The curve follows the model as shown before, which means that the net accesses increase as  $5N/2$ , where  $N$  is the number of processors. The curves for backoff on the barrier variable alone, and backoff on the barrier flag with backoff constant 2, 4, and 8 are also shown (as mentioned before, all our simulated cases of backoff on the barrier flag include first backing-off on the barrier variable.)

Figure 5 corresponds well with our model's prediction of an average 20% reduction of synchronization references due to backing off with information from the barrier variable, i.e., the backoff on the barrier variable gives  $3N/2$  network accesses. Not surprisingly, using binary backoff (or backoff with constants 4 or 8) on the barrier flag made no difference because everyone reaches the barrier at the same time when  $A = 0$ . The backoff on the barrier variable results in each processor spending very little time polling the barrier flag waiting for it to change. For example, for the 64 processor case, a processor on average accessed the network 32 times to get at the barrier variable, 96 times to test the flag before it was set, and 32 times after it was set, for a total of about 160 network accesses. With backoff on the barrier variable this number reduced to roughly 132, a 15% reduction.

Backoff with  $A=1000$  often has a savings greater than the log of the time interval of arrival at the barrier because of reduced interference with the final write request into the flag. This phenomenon also explains the fewer network accesses for backoff with base 8 at  $A=1000$  than at  $A=0$  for 32 processors. However, this savings often comes at the expense of increased processor waiting times.

Figures 6 and 7 correspond to the network accesses by a process for  $A = 100$  and  $A = 1000$  respectively. In Figure 6 for the backoff on the barrier variable we see similar savings as in Figure 5 with  $A = 0$  because the interval  $A$  is still not very big compared to the number of processors. Note, however, the big reductions that the exponential backoffs on the barrier flag gave. With  $A = 100$ , not everyone reaches the barrier flag simultaneously, so the ones who arrive early backoff by some exponential constant rather than continuously polling the barrier flag. In the 16 processor case with a base 4 backoff on the barrier flag, for example, we see a savings of over 90%. In a 64 processor case with a base 8 backoff, the savings are in network accesses is about 60%.

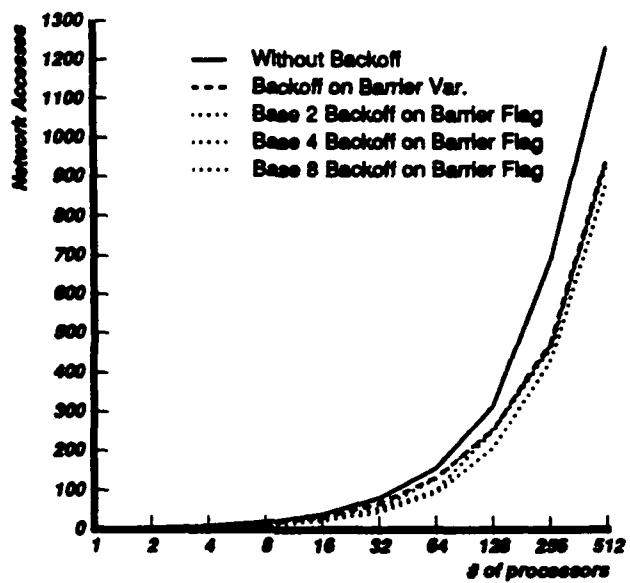


Figure 5: Performance of backoff algorithms for  $A = 0$ .

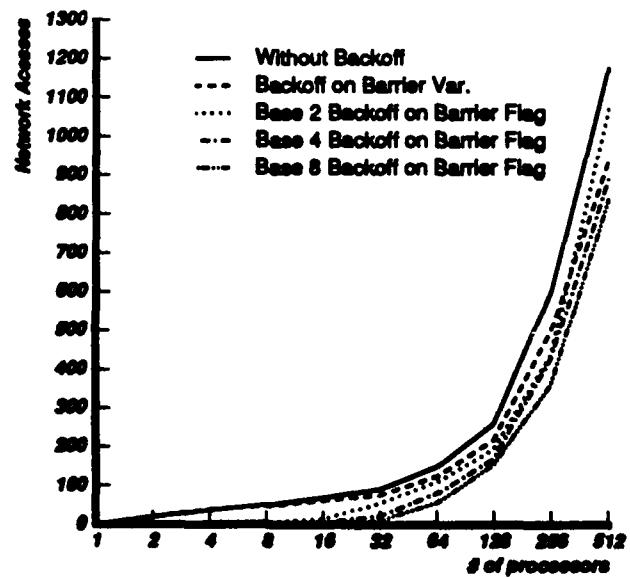


Figure 6: Performance of backoff algorithms for  $A = 100$ .

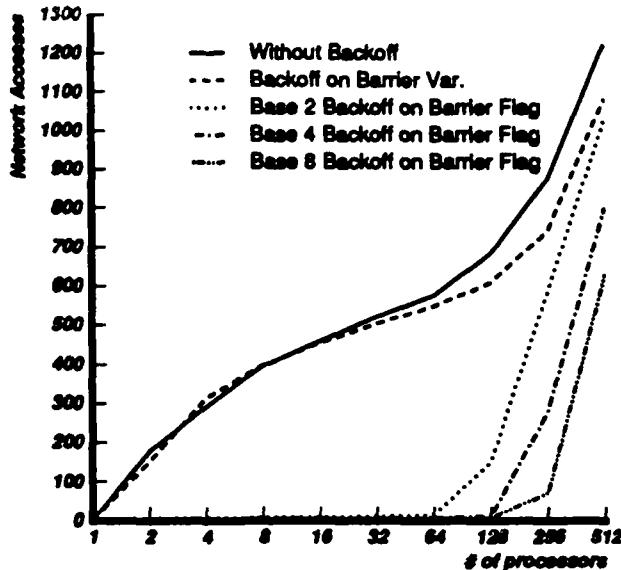


Figure 7: Performance of backoff algorithms for  $A = 1000$ .

The proportional benefit due to backoff decreases as  $N$  increases because contention in the network to access the barrier flag becomes a sizable portion of the network accesses. Recall that an unsuccessful network access in accessing the barrier flag is still counted as a network access. (To reduce these unsuccessful accesses one might use backoff techniques in the network accessing. This issue is discussed later.) For example, in the  $A = 100$  and  $N = 512$  case with base 8 backoff, the reduction in network accesses was only about 30%.

For  $A = 1000$  backoff on the barrier variable once again offers only modest savings. It is interesting to note that for up to 32 processors this scheme offers virtually no savings, because not many processors are contending on the barrier flag. The savings become more significant for larger numbers of processors because the backoff on the barrier variable reduces the length of time that all of the processors spend polling the barrier flag. For 256 processors, for example, backoff on the barrier variable yields about a 15% improvement.

The savings due to exponential backoff on the barrier flag with  $A = 1000$ , however, are quite dramatic. Since the processors potentially have a large interval to poll the barrier flag before everyone arrives, exponentially backing off between testing the flag helps tremendously. In the 16 processor case with a binary backoff on the flag, for example, we see over a 95% savings in network accesses. The 64 processor case offers a similar improvement. This reduction roughly approximates a  $\log_b$  reduction in the number of accesses, where  $b$  is the base used in the exponential backoff.

The small number of network accesses with backoff on the barrier flag for the cases  $A = 0$  and  $N < 8$ ,  $A = 100$  and  $N < 32$ , and  $A = 1000$  and  $N < 128$ , compares reasonably with the network accesses in the bus-based schemes, the broadcast based schemes, or the Hoshino scheme, with no extra hardware or the broadcast requirement. However, when  $A$  is smaller or  $N$  is larger, the backoff schemes tend to do much worse than the schemes that have special hardware support for synchronisation.

It is clear that backoff on the barrier flag is potentially much more beneficial for large  $A$  because most of the network accesses that happen while the processes await the remaining processes to arrive at the barrier can be obviated. These accesses correspond to the first term in the Model 2 equation. Backoff on the barrier variable alone does not impact performance significantly when  $N$  is small compared to  $A$ , but can yield up to a 20% improvement when  $N$  is large.

It is interesting to see that the network accesses increase dramatically for  $N = 128$  ( $A = 1000$ ). It seems that the backoff techniques are not as useful in this case (improvement is less than about 30% for  $N = 256$  and backoff

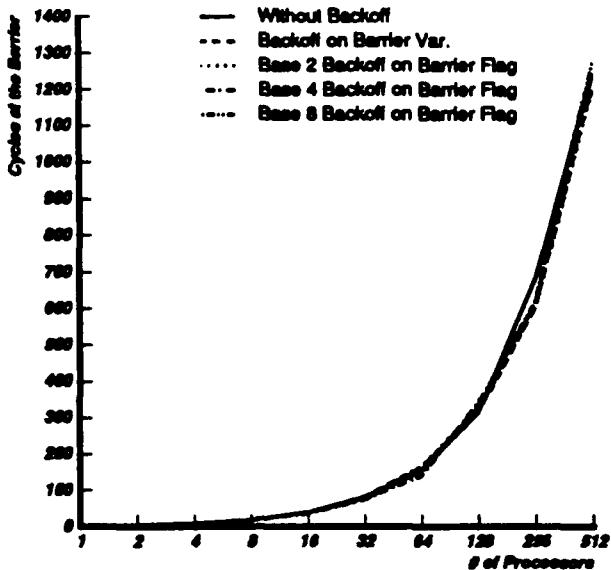


Figure 8: Processor waiting times for backoff algorithms for  $A = 0$ .

with constant 2), although for these cases barrier synchronization is probably inappropriate anyway without some form of distributed software combining [25]. Our backoff methods can still be used on the intermediate nodes of the combining tree. The reason for the sharp increase can be described as follows: When the number of processors is small compared to  $A$ , a process can get access to the barrier flag usually within one network access. However, when the number of processors is not small compared to  $A$ , then a process will suffer contention in trying to access the barrier flag, and contention shows up as repeated network accesses.

In both cases the network accesses can be dramatically reduced for  $N < 128$ . For larger  $N$ , when the contention due to multiple processors simultaneously accessing the barrier increases, the percentage benefits decrease. Note we do nothing about these contention accesses. A method described in the next section will show a method to reduce this problem.

Our simulations show that using a backoff method on both the barrier variable and the barrier flag can yield savings from 20% to over 95% of the network accesses. However, the reduction in network traffic using the backoff methods does not always come for free. Because a backoff method can cause unnecessary processor idle time, we must carefully analyze the delays that these techniques can introduce. The occurrence of delays alone might not be a major cause for alarm, because these delays correspond to the delays suffered by the synchronizing processes alone, and do not affect other processes. The next section addresses these issues.

## 7 Discussion of Tradeoffs

An appropriate backoff constant must be determined by trading off the reduction in network accesses with the potential increase in the number of cycles the CPU spends idling during backoff.

Figures 8 through 10 correspond to the average waiting times for each of the processes for  $A = 0, 100, 1000$  respectively. The waiting time for a process is computed as the number of cycles between first arriving at the barrier to when the process finds the barrier flag set. The graphs denote the four cases shown previously, that is, without backoff, with backoff on the barrier variable and with exponential backoff on the barrier flag with bases 2, 4 and 8.

We see that in all cases binary backoff provides a favorable tradeoff between large reductions in synchronization references and contained increases in wasted CPU cycles. In the sixty-four processor case when  $A = 1000$ ,

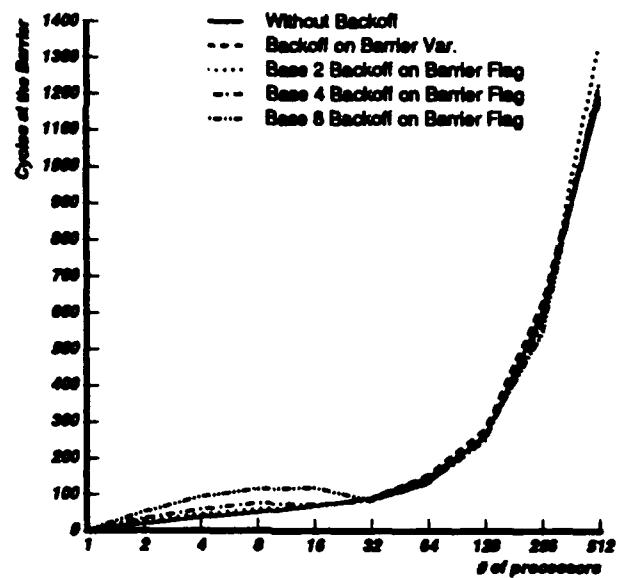


Figure 9: Processor waiting times for backoff algorithms for  $A = 100$ .

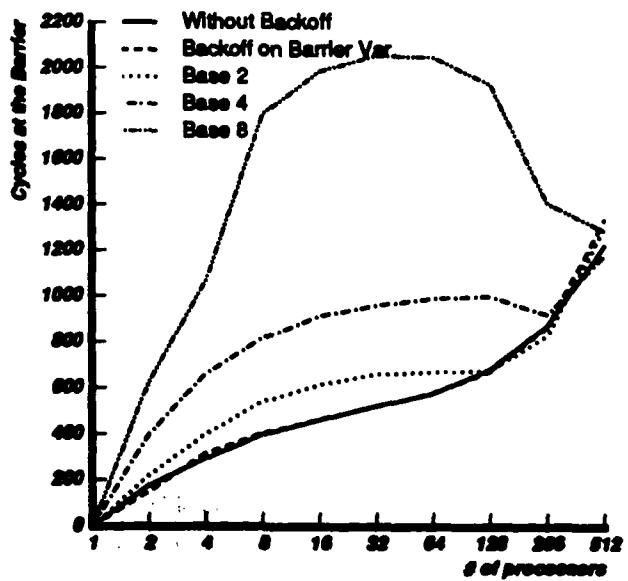


Figure 10: Processor waiting times for backoff algorithms for  $A = 1000$ .

for example, the binary backoff decreased synchronization accesses by 97% while increasing the time spent at the barrier by only 16%.

For  $A = 0$ , and  $A = 100$ , the waiting times for all the four curves are similar because the opportunity for a large backoff time is rare given that all the processes arrive within 100 cycles of each other. The waiting time in these cases is proportional to the number of network accesses, as it is precisely these network accesses that give rise to the delays at the barrier. This intuition is corroborated by the strong resemblance of the curves in Figures 6 and 9.

The average time spent idling can increase dramatically when  $A$  is large because of the possibility of large backoff times. This opportunity is greater for the base 4 and base 8 exponential backoff schemes. As an example, for 64 processors and  $A = 1000$ , the waiting times without backoff and with base 8 exponential backoff on the flag are 576 and 2048 respectively – depicting an increase of over 350% due to backoff. Even in this case, one important benefit is that the barrier accesses are both reduced and spread out uniformly over time.

When the arrival interval  $A$  is much larger than the number of processors, and a high processor utilization is important, one can modify the backoff algorithm as follows. If the backoff amount crosses some preset threshold, then it might be worthwhile to place the process on a queue pending the arrival of the last process. The enqueueing operation incurs a constant overhead that might be unnecessary should the processes arrive within a small interval. Because  $A$  cannot often be determined a priori, such a method of deciding when to put a process to sleep might be promising.

Interestingly, for the  $A = 1000$  case, the average waiting times per processor reach a maximum around 64 processors and then actually decline as  $N$  increases. When the number of processors is small compared to  $A$ , the processors can test the flag without excessive contention with other processes. After each unsuccessful test, they back off, and the backoff time is exponentially related to the number of times they unsuccessfully back off. Because the number of such accesses can be quite large when contention is low and  $A$  is large, there arises the potential for overshooting the point where the flag is set by a large amount. Conversely, when the number of processors is comparable to  $A$  (or greater than  $A$ ), the number of times a process manages to access the barrier flag is small due to contention with other processes. In such cases, the network access count increases, but the average waiting time per processor decreases. Referring to Figures 7 and 10 the decrease in the waiting time for the backoff curves closely corresponds to the increase in network accesses.

## 7.1 Summary

A few general observations can be made at this point. When the number of processors participating in the barrier synchronization is small compared to the time of arrival of the processors, significant reduction in network accesses can be achieved without compromising processor utilization due to backoff waiting for a small backoff base. In such cases, the number of synchronization network accesses is similar to those made in schemes that use special hardware support such as synchronization buses, broadcasts, or global synchronization logic. When the number of processors is large, and if they arrive within a relatively small interval of time, a penalty in either network accesses or processor idle time must be paid. However, depending on the situation, one can be traded for the other.

Our discussion thus far focused on the traffic and the waiting time *during* the execution of the barrier. We can also look at the effect on average traffic with the caveat that such smoothing might tend to make barrier accesses seem less disruptive. We measured the average network data traffic per processor in FFT (assuming separate packet-switched networks for the request and response), excluding synchronization references, to be 0.133 network accesses per cycle. Using results from our simulations of the barriers with  $A = 100$  (roughly approximating the barrier interval  $A$  in FFT with 64 processors) we compute the extra traffic due to barriers when the barrier variable and the barrier flag are not cached. Adding these synchronization references to our base network traffic, the average traffic increases to 0.136 network accesses per cycle (assuming that the base traffic in  $A$  is also 0.133). Now, with a base 8 exponential backoff we find that the average network traffic drops to 0.134. This decrease is significant considering that these savings come from reductions in synchronization references which are effectively hot-spot references. Moreover, we observe in this case that the base 8 exponential backoff also results in a 10 percent decrease in waiting time at the barrier. Both average network traffic and waiting time at synchronizations are reduced using backoff methods for our FFT application.

As a validation of our barrier simulation model, we also compared the average network traffic in FFT when synchronization references are not cached with the average network traffic predicted by our barrier model simulations. The numbers correlated well, with barrier simulations predicting 0.136 net accesses per cycle per processor, while measurements from FFT yielded 0.135.

We analyzed the tradeoff between network accesses and processor idle time due to backoff. In general, reducing the number of network accesses might be more important than reducing the processor idle time because reducing the number of network accesses also reduces the processor idle time because of the reduced contention in the network, and because of decreased competition with the regular network activities of the other processors not involved in the barrier.

## 8 Optimizations and Extensions

This paper focused on the effect of adaptive backoff techniques on barrier synchronization. The same methods can be applied in several other cases. For example, this technique can be applied to processors waiting on a resource. Processors waiting to access a resource can backoff testing the resource by an amount proportional to the number of processors waiting. Adaptive techniques will likely perform much better in this situation than with barrier synchronizations because the amount of time a processor has to wait at a resource is directly proportional to the number of processors waiting (with the constant of the proportion being the average amount of time the resource is held by each processor). In a barrier situation, the amount of time a processor has to wait at the barrier flag is not necessarily directly proportional to the number of processors which have reached the barrier.

Another similar method that can reduce contention in unbuffered circuit-switched networks is to use adaptive backoff methods for network accesses also. If a network access suffers a collision, instead of resubmitting the request immediately, one can backoff some amount first. This backoff amount can be determined in one of several ways:

(1) For example, a network supplied status byte can be used to determine the stage at which the collision occurred. The backoff amount can be proportional to the network depth traversed by the message. The rationale for this choice is that the deeper a message travels, the greater the network resource that it ties up in its unsuccessful attempt. Conversely, if a collision occurs within a few stages of travel into the network, the access can be resubmitted sooner as the network resources tied up will be smaller.

(2) An argument for making the backoff amount inversely proportional to the network depth traversed can also be made. The deeper a message travels before colliding, the less congested the network is expected to be, and so the access can be retried sooner. Simulations can be used to study the tradeoffs involved in these two opposing arguments and suggest a practical backoff algorithm.

(3) On a collision, a network access might wait some constant time proportional to the average round trip time to memory through the network before resubmitting the request.

(4) The number of previous unsuccessful tries can be used as a parameter to an exponential backoff algorithm.

(5) In a packet-switched network, Scott and Sohi [20] make use of the state information found in the queues at the memory modules to signal processors to stop making requests in congested situations. This state information could also be used to have the processors back off sending requests by some time proportional to the length of the queue.

As we mentioned before, the adaptive backoff techniques that we evaluated do not require special hardware support. The synchronization software that determines which backoff method is used can be designed in one of several ways. One can be conservative and use a simple adaptive backoff on the barrier variable and a binary backoff on the barrier flag. The programmer can write the algorithms into the synchronization macros or routines from a knowledge of the application. The compiler can determine appropriate code sequences for the barrier synchronizations based on expected behavior of loops and the amount of visible parallelism. One can get more venturesome by using profiling to determine the temporal behavior of the application and the number of processors participating in the synchronization and pass this information on to the compiler for further optimization. One case where such information might be useful is in determining when to (or whether to) queue a process to await a signal when the barrier flag is set rather than spinning on the network.

## 9 Conclusions

Network bandwidth is a precious resource in large-scale shared memory multiprocessors. In this paper we present a group of adaptive synchronization techniques aimed at reducing the number of network accesses due to synchronizations. We model adaptive techniques for barrier synchronizations and show that in some cases these techniques can achieve dramatic savings at minimal extra cost, while in other situations network accesses can be reduced while trading-off processor utilization of synchronizing processors. These techniques are implemented in software, and they can be optimized for varying applications.

The central idea behind an adaptive synchronization technique is to make use of information available from synchronization state and from past history to reduce the number of idle synchronization spins. The general technique is useful for barrier synchronizations as well as other situations such as reducing accesses made by processors waiting on a resource or reducing contention in unbuffered circuit-switched networks.

## 10 Acknowledgements

We thank Kimmung So, Harold Stone, and Prabhakar Raghavan for their insights contributed during discussions on many of the ideas presented in this paper. Kimmung So aided us greatly in obtaining traces from PSIMUL. We also thank Pat Teller from NYU who provided the SIMPLE and WEATHER programs parallelized under the Epex environment. The research reported in this paper was funded by the Defense Advanced Research Projects Agency under contract # N00014-87-K-0825, and by IBM under a joint research program.

## A Tracing Methodology

The multiprocessor traces we used for our simulations were generated using a "post-mortem scheduling" technique in which a multiprocessor trace is created from a memory reference trace of a uniprocessor execution of a parallel application. Key to the scheme is that the uniprocessor execution trace include information about synchronization events in the code. Using this record of synchronization events, a scheduler can schedule tasks from the uniprocessor execution trace into a multiprocessor trace in which the synchronization sections are simulated.

This methodology can be used for a variety of programming paradigms. The two applications we traced are both written in Epex/Fortran using the Single-Program-Multiple-Data (SPMD) computational model [6]. In this model all processes are created at the beginning of the program and execute the same program. Though all processes are executing the same program, synchronization constructs embedded in the code dynamically determine which sections of the program processors execute. The SPMD model for Epex/Fortran contains serial and parallel sections along with replicate sections, which are executed by all processors. We use this model in the FFT, SIMPLE and WEATHER applications because it is a good method by which to exploit the parallelism in these scientific applications without making major changes (likely modifying the fundamental algorithms used) to the already existing uniprocessor code.

The post-mortem scheduler simulates synchronization events in the application using some prescribed synchronization implementation. We simulate fetch-and-adds (F&A), a synchronization primitive used to exclusively update a location in memory, with an atomic read-modify-write operation. In EPEX/FORTRAN, synchronization constructs at the beginning of parallel and serial sections perform F&As on shared variables to determine task assignments to processes. Barriers and waits at the end of loops and serial sections are simulated by arriving processors first incrementing a shared variable through a F&A and then polling a barrier flag until it is set by the last arriving processor.

The uniprocessor memory reference trace with synchronization information was produced by PSIMUL [21], a multiprocessor simulator. PSIMUL generates IBM S/370 memory reference traces and has the capability of marking down into the trace the type of synchronization constructs it traverses while tracing the application. Our scheduler simulates a parallel execution of this trace, assigning processors references from the trace on a round-robin basis. We assume that processors make a memory reference every cycle, which is an approximation because the S/370 instruction set contains register-to-register instructions.

The Fast Fourier Transform (FFT) [4] application, written at IBM, is a parallelized version of a Radix-2 FFT computation in two variables on a random array of complex numbers. Since we used a problem size of 128, the parallel loops working on the 128x128 matrix contained 128-way parallelism. This provided for an even distribution of work among processors for the 64 processor simulations. We traced two passes of the TF2 routine, which computes the FFT, through the matrix, first by rows and then by columns. FFT is an example of a highly uniform parallel application in which processors execute parallel loop iterations of approximately equal length and arrive at barriers within close intervals.

The SIMPLE code models hydrodynamic and thermal behavior of fluids in two dimensions [5]. Finite difference methods are used to solve the equations of inviscid compressible hydrodynamics and simple heat conduction. The problem is formulated on an NxN mesh. Once again, we used a problem size of 128, but many of the parallel sections in SIMPLE do not contain fully 128-way parallelism. The resulting distribution of work among the 64 processors in our simulations is uneven. Sixty-four processors is, however, the optimal number of processors to execute this application, given the problem size. Another important difference in this application from FFT is that SIMPLE contains a number of small and large parallel loops (20 in all) rather than the few large parallel loops that FFT contains. SIMPLE also contains many small serial sections (5) in which one processor executes the serial section while all the rest wait at the bottom. The resulting difference is that SIMPLE contains far more synchronization activity than FFT. Parallel loop iteration lengths in SIMPLE vary occasionally, also contributing to more synchronization accesses due to more processor waiting at the end of a parallel loops with uneven loop iterations. SIMPLE would be representative of a typical application which allows neither worst-case, nor best-case performance giving our SPMD computational model.

The WEATHER code forecasts the weather by modeling the state of the atmosphere as described by the NASA GLAS/GISS fourth order general circulation model of a three-dimensional atmosphere [11]. The algorithm breaks the atmosphere down into a three-dimensional grid encircling the globe and computes the value of several interrelated state variables using finite difference methods. In the model simulated by WEATHER, the atmosphere was represented by nine regions of fixed altitude and a grid uniformly spread across longitude and latitude on each layer. In the runs we traced, the grid was 108 by 72. Parallel sections of the COMP1 routine, which calculates horizontal and vertical advection differences in the atmosphere, were traced. The load-balancing in this application is far worse than in FFT and SIMPLE, given that it was simulated with 64 processors. Since the parallelism is derived by simultaneously working on rows/columns of the atmosphere grid, and the dimensions of the grid are not multiples of 64, many processors are forced to idle in parallel sections which are followed by barriers. Fifty-four processors would be a more optimal number of processors to execute this application with the problem size used. Thus the load balancing in our three applications showed a wide range.

## References

- [1] Norman Abramson. The ALOHA System – Another alternative for computer communications. In *Proc. of the 1977 Fall Joint Computer Conf.*, pages 281–285, 1977.
- [2] Anant Agarwal, Richard Simoni, John Hennessy, and Mark Horowitz. An Evaluation of Directory Schemes for Cache Coherence. In *Proceedings of the 15th International Symposium on Computer Architecture*, IEEE, New York, June 1988.
- [3] Lucien M. Censier and Paul Feautrier. A New Solution to Coherence Problems in Multicache Systems. *IEEE Transactions on Computers*, C-27(12):1112–1118, December 1978.
- [4] J. W. Cooley and J. W. Tukey. An Algorithm for the Machine Calculation of Complex Fourier Series. *Math. Comput.*, 19:297–301, April 1965.
- [5] W. P. Crowley, C. P. Hendrickson, and T. E. Rudy. *The Simple Code*. Technical Report, Lawrence Livermore Laboratory, February 1978.
- [6] F. Damera-Rogers, D. A. George, V. A. Norton, and G. F. Pfister. *Single-Program-Multiple-Data Computational Model for EPEX/FORTRAN*. Technical Report RC 11552 (55212), IBM T. J. Watson Research Center, Yorktown Heights, November 1986.

- [7] Daniel Gajski, David Kuck, Duncan Lawrie, and Ahmed Saleh. Cedar - A Large Scale Multiprocessor. In *International Conference on Parallel Processing*, pages 524-529, August 1983.
- [8] James R. Goodman. Using Cache Memory to Reduce Processor-Memory Traffic. In *Proceedings of the 10th Annual Symposium on Computer Architecture*, pages 124-131, IEEE, New York, June 1983.
- [9] A. Gottlieb, R. Grishman, C. P. Kruskal, K. P. McAuliffe, L. Rudolph, and M. Snir. The NYU Ultracomputer - Designing a MIMD Shared-Memory Parallel Machine. *IEEE Transactions on Computers*, C-32(2):175-189, February 1983.
- [10] Tsutomu Hoshino. *PAX Computer. High-Speed Parallel Processing and Scientific Computing*. Addison Wesley, Reading Mass., 1989. Editor Harold S. Stone.
- [11] Eugenia Kalnay-Rivas and David Hoitsma. *Documentation of the Fourth Order Band Model*. Technical Report, NASA Modeling and Simulation Facility Laboratory for Atmospheric Science, NASA/Goddard Space Flight Center, Greenbelt, MD, 1979.
- [12] L. Kleinrock and Y. Yemini. An Optimal Adaptive Scheme for Multiple Access Broadcast Communication. In *Proceedings ICC*, June 1978.
- [13] S. S. Lam. A Carrier Sense Multiple Access Protocol for Local Networks. *Computer Networks*, 4(1), January 1980.
- [14] S. S. Lam and L. Kleinrock. Packet Switching in a Multiaccess Broadcast Channel: Dynamic Control Procedures. *IEEE Transactions on Computers*, C-23, Sept. 1975.
- [15] E. L. Lusk and R. A. Overbeek. *Implementation of Monitors with Macros: A Programming Aid for the HEP and other Parallel Processors*. Technical Report ANL-83-97, Argonne National Laboratory, Argonne, Illinois, December 1983.
- [16] R. Metcalfe and D. Boggs. Ethernet: Distributed Packet Switching for Local Computer Networks. *Communications of the ACM*, 19(7), July 1976.
- [17] Janak H. Patel. Analysis of Multiprocessors with Private Cache Memories. *IEEE Transactions on Computers*, C-31(4):296-304, April 1982.
- [18] G. F. Pfister, W. C. Brantley, D. A. George, S. L. Harvey, W. J. Kleinfelder, K. P. McAuliffe, E. A. Melton, A. Norton, and J. Weiss. The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture. In *Proceedings ICPP*, pages 764-771, August 1985.
- [19] G. F. Pfister and V. A. Norton. 'Hotspot' Contention and Combining in Multistage Interconnection Networks. *IEEE Transactions on Computers*, C-34(10), October 1985.
- [20] Steven Scott and Gurindar Sohi. Using Feedback to Control Tree Saturation In Multistage Interconnection Networks. In *Proceedings 16th Annual Int'l Symp. on Computer Architecture*, June 1989.
- [21] K. So, F. Damera-Rogers, D. A. George, V. A. Norton, and G. F. Pfister. *PSIMUL - A System for Parallel Simulation of Parallel Systems*. Technical Report RC 11674 (58502), IBM T. J. Watson Research Center, Yorktown Heights, November 1987.
- [22] Peiyi Tang and Pen-Chung Yew. Processor Self-scheduling for Multiple-Nested Parallel Loops. In *Proceedings of the 1986 International Conference on Parallel Processing*, pages 528-535, August 1986.
- [23] Charles P. Thacker and Lawrence C. Stewart. Firefly: a Multiprocessor Workstation. In *Proceedings of ASPLOS II*, pages 164-172, October 1987.
- [24] Wolf-Dietrich Weber and Anoop Gupta. Analysis of Cache Invalidation Patterns in Multiprocessors. In *Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS III)*, April 1989.
- [25] P.-C. Yew, N.-F. Tseng, and D. H. Lawrie. Distributed Hot-Spot Addressing in Large-Scale Multiprocessors. *IEEE Transactions on Computers*, C-36(14):388-395, April 1987.